AD-A252 470

ITATION PAGE

Form Approved OPM No.

Pu... an... su... 22...

...erage 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering ...ion. Send comments regarding this burden estimate or any other aspect of this collection of information, including ...e, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA ...of Management and Budget, Washington, DC 20503.

1.	2. REPORT	3. REPORT TYPE AND DATES Final: 18 Mar 1992 01 Jun 1993

4. TITLE AND Validation Summary Report: Tartan, Inc., Tartan Ada SPARC 1750a version 4.2, Sun SPARCstation/ELC (Host) to Fairchild F9450 on a SBC-50 board (MIL-STD 1870a)(Target) 920313I1.11245	5. FUNDING 2
6. IABG-AVF Ottobrunn, Federal Republic of Germany	
7. PERFORMING ORGANIZATION NAME(S) AND IABG-AVF, Industrieanlagen-Betriebsgeselschaft Dept. SZT/ Einsteinstrasse 20 D-8012 Ottobrunn FEDERAL REPUBLIC OF GERMANY	8. PERFORMING ORGANIZATION IABG-VSR 84
9. SPONSORING/MONITORING AGENCY NAME(S) AND Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081	10. SPONSORING/MONITORING AGENCY
11. SUPPLEMENTARY	
12a. DISTRIBUTION/AVAILABILITY Approved for public release; distribution unlimited.	12b. DISTRIBUTION

DTIC ELECTE JUL 0 1 1992 S A D

13. (Maximum 200

Tartan, Inc., Tartan Ada SPARC 1750a version 4.2, Sun SPARCstation/ELC (Host) to Fairchild F9450 on a SBC-50 board (MIL-STD 1870a)(Target), AVCV 1.11.

92 ... 3

92-17196

14. SUBJECT Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A,	15. NUMBER OF 16. PRICE

17. SECURITY CLASSIFICATION UNCLASSIFIED	18. SECURITY UNCLASSIFED	19. SECURITY CLASSIFICATION UNCLASSIFIED	20. LIMITATION OF

NSN

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std.

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 13 March, 1992.

Compiler Name and Version:	**Tartan Ada SPARC 1750a version 4.2**
Host Computer System:	**Sun SPARCstation/ELC under SunOS Version 4.1.1**
Target Computer System:	**Fairchild F9450 on a SBC-50 board (MIL-STD-1750a, bare machine)**

See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 920313I1.11245 is awarded to Tartan, Inc. This certificate expires 24 months after ANSI approval of MIL-STD 1815B.

This report has been reviewed and is approved.

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

AVF Control Number: IABG-VSR 84
18 March, 1992

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 920313I1.11245
Tartan, Inc.
Tartan Ada SPARC 1750a version 4.2
Sun SPARCstation/ELC =>
Fairchild F9450 on a SBC-50 board
(MIL-STD 1750a)

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 13 March, 1992.

Compiler Name and Version:	**Tartan Ada SPARC 1750a version 4.2**
Host Computer System:	**Sun SPARCstation/ELC under SunOS Version 4.1.1**
Target Computer System:	**Fairchild F9450 on a SBC-50 board (MIL-STD-1750a, bare machine)**

See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 920313I1.11245 is awarded to Tartan, Inc. This certificate expires 24 months after ANSI approval of MIL-STD 1815B.

This report has been reviewed and is approved.

Michael Tonndorf

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: Tartan, Inc.

Certificate Awardee: Tartan, Inc.

Ada Validation Facility: IABG mbH

ACVC Version: 1.11

Ada Implementation:

Ada Compiler Name and Version: Tartan Ada SPARC 1750a version 4.2

Host Computer System: SPARC Station/ELC SunOS version 4.1.1

Target Computer System: Fairchild F9450 (MIL-STD-1750a,bare machine)

Declaration:

I, the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

[signature]
Customer Signature

March 16 1992
Date

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 USE OF THIS VALIDATION SUMMARY REPORT 1-1
1.2 REFERENCES. 1-1
1.3 ACVC TEST CLASSES 1-2
1.4 DEFINITION OF TERMS 1-2

CHAPTER 2 IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS 2-1
2.2 INAPPLICABLE TESTS. 2-1
2.3 TEST MODIFICATIONS. 2-3

CHAPTER 3 PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT 3-1
3.2 SUMMARY OF TEST RESULTS 3-1
3.3 TEST EXECUTION. 3-2

APPENDIX A MACRO PARAMETERS

APPENDIX B COMPILATION SYSTEM OPTIONS

APPENDIX C APPENDIX F OF THE Ada STANDARD

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint	The part of the certification body which provides policy and

Program Office (AJPO)	guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].

Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 02 August 1991.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 285 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

The following 21 tests check for the predefined type SHORT_INTEGER; for this implementation, there is no such type:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

A35801E checks that FLOAT'FIRST..FLOAT'LAST may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (See section 2.3.)

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45536A, C46013B, C46031B, C46033B, and C46034B contain length clauses that specify values for 'SMALL that are not powers of two or ten; this implementation does not support such values for 'SMALL.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

D64005G uses 17 levels of recursive procedure calls nesting; this level of nesting for procedure calls exceeds the capacity of the compiler.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C instantiate generic units before their bodies are compiled; this implementation creates a dependence on generic units as allowed by AI-0408 & AI-0506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (see 2.3.)

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

The following 264 tests check operations on sequential, text, and direct access files; this implementation does not support external files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A expect that NAME_ERROR is raised when an attempt is made to create a file with an illegal name; this implementation does not support the creation of external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see Section 1.3) were required for 106 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24007A	B24009A	B25002B	B32201A	B33204A
B33205A	B35701A	B36171A	B36201A	B37101A	B37102A
B37201A	B37202A	B37203A	B37302A	B38003A	B38003B
B38008A	B38008B	B38009A	B38009B	B38103A	B38103B
B38103C	B38103D	B38103E	B43202C	B44002A	B48002A
B48002B	B48002D	B48002E	B48002G	B48003E	B49003A
B49005A	B49006A	B49006B	B49007A	B49007B	B49009A
B4A010C	B54A20A	B54A25A	B58002A	B58002B	B59001A
B59001C	B59001I	B62006C	B67001A	B67001B	B67001C
B67001D	B74103E	B74104A	B74307B	B83E01A	B85007C
B85008G	B85008H	B91004A	B91005A	B95003A	B95007B
B95031A	B95074E	BA1001A	BC1002A	BC1109A	BC1109C
BC1206A	BC2001E	BC3005B	BD2A06A	BD2B03A	BD2D03A
BD4003A	BD4006A	BD8003A			

E28002B was graded inapplicable by Evaluation and Test Modification as directed by the AVO. This test checks that pragmas may have unresolvable arguments, and it includes a check that pragma LIST has the required effect; but, for this implementation, pragma LIST has no effect if the compilation results in errors or warnings, which is the case when the test

is processed without modification. This test was also processed with the pragmas at lines 46, 58, 70 and 71 commented out so that pragma LIST had effect.

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO. The compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. LRM 3.5.7:12).

Tests C45524A..E (5 tests) were graded passed by Test Modification as directed by the AVO. These tests expect that a repeated division will result in zero; but the Ada standard only requires that the result lie in the smallest safe interval. Thus, the tests were modified to check that the result was within the smallest safe interval by adding the following code after line 141; the modified tests were passed:

```
ELSIF VAL <= F'SAFE_SMALL THEN COMMENT ("UNDERFLOW SEEMS GRADUAL");
```

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package report's body, and thus the packages' calls to function Report.Ident_Int at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

B83E01B was graded passed by Evaluation Modification as directed by the AVO. This test checks that a generic subprogram's formal parameter names (i.e. both generic and subprogram formal parameter names) must be distinct; the duplicated names within the generic declarations are marked as errors, whereas their recurrences in the subprogram bodies are marked as "optional" errors--except for the case at line 122, which is marked as an error. This implementation does not additionally flag the errors in the bodies and thus the expected error at line 122 is not flagged. The AVO ruled that the implementation's behavior was acceptable and that the test need not be split (such a split would simply duplicate the case in B83E01A at line 15).

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C were graded inapplicable by Evaluation Modification as directed by the AVO. These tests instantiate generic units before those units' bodies are compiled; this implementation creates dependences as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete, and the objectives of these tests cannot be met.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by compiling the seperate files in the following order (to allow re-compilation of obsolete units), and all intended errors were then detected by the compiler:

BC3204C: C0, C1, C2, C3M, C4, C5, C6, C3M

BC3205D: D0, D1M, D2, D1M

BC3204D and BC3205C were graded passed by Test Modification as directed by the AVO. These tests are similar to BC3204C and BC3205D above, except that all compilation units are contained in a single compilation. For these two tests, a copy of the main procedure (which later units make obsolete) was appended to the tests; all expected errors were then detected.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-ten value as small for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

AD9001B and AD9004A were graded passed by Processing Modification as directed by the AVO. These tests check that various subprograms may be interfaced to external routines (and hence have no Ada bodies). This implementation requires that a file specification exists for the foreign subprogram bodies. The following command was issued to the Librarian to inform it that the foreign bodies will be supplied at link time (as the bodies are not actually needed by the program, this command alone is sufficient):

```
interface -sys -L=library ad9001b & ad9004a
```

CE2103A, CE2103B and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical information about this Ada implementation, contact:

Mr Ron Duursma
Director of Ada Products
Tartan, Inc.
300 Oxford Drive
Monroeville, PA 15146
USA
Tel. (412) 856-3600

For sales information about this Ada implementation, contact:

Ms. Marlyse Bennett
Director of Sales
Tartan, Inc.
12110 Sunset Hills Road
Suite 450
Reston, VA 22090
USA
Tel. (703) 715-3044

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a

floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3460	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	66	
d) Non-Processed I/O Tests	264	
e) Non-Processed Floating-Point Precision Tests	285	
f) Total Number of Inapplicable Tests	615	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

A magnetic data cartridge containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic data cartridge were loaded directly onto the host computer.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link, an RS232 Interface, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were for compiling:

-f forces the compiler to accept an attempt to compile a unit imported from another library which is normally prohibited.

-c suppresses the creation of a registered copy of the source code in the library directory for use by the REMAKE and MAKE subcommands.

-La forces a listing to be produced, default is to only produce a listing when an error occurs.

No explicit linker options were used.

Test output, compiler and linker listings, and job logs were captured on magnetic data cartridge and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	240
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	16
$ALIGNMENT	1
$COUNT_LAST	32766
$DEFAULT_MEM_SIZE	65536
$DEFAULT_STOR_UNIT	16
$DEFAULT_SYS_NAME	MIL_STD_1750A
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.ADDRESS'(16#000_000D#)
$ENTRY_ADDRESS1	SYSTEM.ADDRESS'(16#000_000E#)
$ENTRY_ADDRESS2	SYSTEM.ADDRESS'(16#000_000F#)
$FIELD_LAST	240
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.7014111E+38
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.7014111E+38
$HIGH_PRIORITY	200
$ILLEGAL_EXTERNAL_FILE_NAME1	ILLEGAL_EXTERNAL_FILE_NAME1
$ILLEGAL_EXTERNAL_FILE_NAME2	ILLEGAL_EXTERNAL_FILE_NAME2
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1

$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006F1.TST")
$INTEGER_FIRST	-32768
$INTEGER_LAST	32767
$INTEGER_LAST_PLUS_1	32768
$INTERFACE_LANGUAGE	CLink
$LESS_THAN_DURATION	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST	-131_073.0
$LINE_TERMINATOR	' '
$LOW_PRIORITY	10
$MACHINE_CODE_STATEMENT	Two_Opnds'(LR,(R_am,R0),(R_am,R1));
$MACHINE_CODE_TYPE	Instruction_Mnemonic
$MANTISSA_DOC	31
$MAX_DIGITS	9
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648
$MIN_INT	-2147483648
$NAME	NO_SUCH_TYPE_AVAILABLE
$NAME_LIST	MIL_STD_1750A
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	1048576
$NEW_STOR_UNIT	16
$NEW_SYS_NAME	MIL_STD_1750A
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	record Operation: Instruction_Mnemonic; Operand_1: Operand; end record;
$RECORD_NAME	Two_Opnds
$TASK_SIZE	16
$TASK_STORAGE_SIZE	1024
$TICK	0.0001
$VARIABLE_ADDRESS	SYSTEM.ADDRESS'(16#0000_0004#)
$VARIABLE_ADDRESS1	SYSTEM.ADDRESS'(16#0000_0005#)
$VARIABLE_ADDRESS2	SYSTEM.ADDRESS'(16#0000_0006#)

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

Chapter 4
Compiling Ada Programs

The `tada1750a` command is used to compile and assemble Ada compilation units.

4.1. *THE* `tada1750a` *COMMAND FORMAT*

The `tada1750a` command has this format:

`% tada1750a [`*option...*`]` *file...* `[`*option...*`]`

Arguments that start with a hyphen are interpreted as options; otherwise, they represent filenames. There must be at least one filename, but there need not be any options. Options and filenames may appear in any order, and all options apply to all filenames. For an explanation of the available options, see Section 4.2.

If a source file does not reside in the directory in which the compilation takes place, *file* must include a path sufficient to locate the file. It is recommended that only one compilation unit be placed in a file.

If no extension is supplied with the file name, a default extension of `.ada` will be supplied by the compiler.

Files are processed in the order in which they appear on the command line. The compiler sequentially processes all compilation units in each file. Upon successful compilation of a unit:

- The library database `Lbrry.Db` is updated with the new compilation time and any new dependencies.
- One or more separate compilation files and/or object files are generated.

If no errors are detected in a compilation unit, `tada1750a` produces an object module and updates the library. If any error is detected, no object code file is produced, a source listing is produced, and no library entry is made for that compilation unit. If warnings are generated, both an object code file and a source listing are produced. For further details about the process of updating the library, files generated, replacement of existing files, and possible error conditions, see Sections 4.3 through 4.5.

The output from `tada1750a` is a file of type `.stof` or `.tof`, for a specification or a body unit respectively, containing object code. Some other files are generated as well. See Section 4.4 for a list of extensions of files that may be generated.

The compiler is capable of limiting the number of library units that become obsolete by recognizing *refinements*. A library unit is a refinement of its previously compiled version if the only changes that were made are:

- Addition or deletion of comments
- Addition of subprogram specifications after the last declarative item in the previous version.

An option is required to cause the compiler to detect refinements. When a refinement is detected by the compiler, dependent units are not marked as obsolete.

4.2. *OPTIONS*

Command line options indicate special actions to be performed by the compiler or special output file properties.

The following command line options may be used:

	$n = 3$	**Time** - Performs level 2 optimizations plus inline expansion of subprogram calls which the optimizer decides are profitable to expand (from an execution time perspective). Other optimizations which improve execution time at a cost to image size are performed only at this level.
	$n = 4$	**Space** - Performs those optimizations which usually produce the smallest code, often at the expense of speed. Please note that this optimization level may not always produce the smallest code. Under certain conditions another level may produce smaller code.

`-p` Extracts syntactically correct compilation unit source from the parsed file and loads this file into the library as a parsed unit. Parsed units are, by definition, inconsistent. This switch allows users to load units into the library without regard to correct compilation order. The command `remakecu` is used subsequently to reorder the compilation units in the correct sequence. See Section 9.2.5 for a more complete description of this command.

`-r` *Data on this switch is provided for* **information only**. This switch is used exclusively by the librarian to notify the compiler that the source undergoing compilation is an internal source file. The switch causes the compiler to retain old external source file information. This switch should be used only by the librarian and command files created by the librarian. See Section 3.6.1.

`-S[ACDEILORSZ]` Suppress the given set of checks:

A	ACCESS_CHECK
C	CONSTRAINT_CHECK
D	DISCRIMINANT_CHECK
E	ELABORATION_CHECK
I	INDEX_CHECK
L	LENGTH_CHECK
O	OVERFLOW_CHECK
R	RANGE_CHECK
S	STORAGE_CHECK
Z	"ZERO" DIVISION_CHECK

The `-S` option has the same effect as an equivalent pragma `SUPPRESS` applied to the source file. If the source program also contains a pragma `SUPPRESS`, then a given check is suppressed if either the pragma or the switch specifies it; that is, the effect of a pragma `SUPPRESS` cannot be negated with the command line option. See LRM 11.7 for further details. Supplying the `-S` option significantly decreases the size and execution time of the compiled code. Examples are:

`-SOZ`	Suppress `OVERFLOW_CHECK` and `"ZERO"DIVISION_CHECK`.
`-S`	Suppress all checks.
`-SC`	Suppress `CONSTRAINT_ERROR`, equivalent to `-SADILR`. (Note that `-SC` is upward compatible with Version 2.0)

`-s` Examines units for syntax errors, then stops compilation without entering a unit in the library.

`-v` Print out compiler phase names. The compiler prints out a short description of each compilation phase in progress.

`-w` Suppress warning messages.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, are outlined below for convenience.

```
package STANDARD is

     ...

   type INTEGER is range -32768 .. 32767;
   type LONG_INTEGER is range -2147483648 .. 2147483647;

   type FLOAT is digits 6 range -16#0.8000_00#e+32 .. 16#0.7fff_ff#e+32;

   type LONG_FLOAT is digits 9 range -16#0.8000_0000_00#e+32 ..
                                      16#0.7fff_ffff_ff#e+32;

   type DURATION is delta 0.0001 range -86400.0 .. 86400.0;

     ...

end STANDARD;
```

Chapter 5
Appendix F to MIL-STD-1815A

This chapter contains the required Appendix F to the LRM which is *Military Standard, Ada Programming Language*, ANSI/MIL-STD-1815A (American National Standards Institute, Inc., February 17, 1983).

5.1. PRAGMAS

5.1.1. Predefined Pragmas

This section summarizes the effects of and restrictions on predefined pragmas.

- Access collections are not subject to automatic storage reclamation so pragma `CONTROLLED` has no effect. Space deallocated by means of `UNCHECKED_DEALLOCATION` will be reused by the allocation of new objects.
- Pragma `ELABORATE` is supported.
- Pragma `INLINE` is supported.
- Pragma `INTERFACE` is supported. It is assumed that the foreign code interfaced adheres to Tartan Ada calling conventions as well as Tartan Ada parameter passing mechanisms. Any other `Language_Name` will be accepted, but ignored, and the default will be used.
- Pragma `LIST` is supported but has the intended effect only if the command line option `-La` was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma `MEMORY_SIZE` is supported. See Section 5.1.3.
- Pragma `OPTIMIZE` is supported except when at the outer level (that is, in a package specification or body).
- Pragma `PACK` is supported.
- Pragma `PAGE` is supported but has the intended effect only if the command line option `-La` was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma `PRIORITY` is supported.
- Pragma `STORAGE_UNIT` is accepted but no value other than that specified in Package `SYSTEM` (Section 5.3) is allowed.
- Pragma `SHARED` is not supported.
- Pragma `SUPPRESS` is supported.
- Pragma `SYSTEM_NAME` is accepted but no value other than that specified in Package `SYSTEM` (Section 5.3) is allowed.

5.1.2. Implementation-Defined Pragmas

Implementation-defined pragmas provided by Tartan are described in the following sections.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma FOREIGN_BODY. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

Pragma LINKAGE_NAME should be used for all declarations in the package, including any declarations in a nested package specification to be sure that there are no conflicting link names. If pragma LINKAGE_NAME is not used, the cross-reference qualifier, -x, (see Section 4.2) should be used when invoking the compiler and the resulting cross-reference table of linknames inspected to identify the linknames assigned by the compiler and determine that there are no conflicting linknames (see also Section 4.6). In the following example, we want to call a function plmn which computes polynomials and is written in assembler.

```
package MATH_FUNCTIONS is
  pragma FOREIGN_BODY ("assembler");
  function POLYNOMIAL  (X:INTEGER) return INTEGER;
        -- Ada spec matching the assembler routine
  pragma LINKAGE_NAME (POLYNOMIAL, "plmn");
        -- Force compiler to use name "plmn" when referring to this
        -- function
end MATH_FUNCTIONS;

with MATH_FUNCTIONS; use MATH_FUNCTIONS;
procedure MAIN is
  X:INTEGER := POLYNOMIAL(10);
        -- Will generate a call to "plmn"
  begin ...
end MAIN;
```

To compile, link and run the above program, you do the following steps:

1. Compile MATH_FUNCTIONS
2. Compile MAIN
3. Provide the object module (for example, math.tof) containing the compiled "C" code for plmn, converted to Tartan Object File Format (TOFF) using the its_to_toff utility (See *Object File Utilities*, Chapter 4)
4. Issue the command:

   ```
   % adalib1750a foreign math_functions math.tof
   ```

5. Issue the command:

   ```
   % adalib1750a link main
   ```

Without Step 4, an attempt to link will produce an error message informing you of a missing package body for MATH_FUNCTIONS.

Using an Ada body from another Ada program library. The user may compile a body written in Ada for a specification into the library, regardless of the language specified in the pragma contained in the specification. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a foreign body with an Ada body without recompiling the specification.

The user can either compile an Ada body into the library, or use the command adalib1750a foreign (see Sections 3.3.3 and 9.5.7) to use an Ada body from another library. The Ada body from another library must have been compiled under an identical specification. The pragma LINKAGE_NAME must have been applied to all entities declared in the specification. The only way to specify the linkname for the elaboration routine of an Ada body is with the pragma FOREIGN_BODY.

5.4.2.1. Size Specifications for Types

The rules and restrictions for size specifications applied to types of various classes are described below.

The following principle rules apply:

1. The size is specified in bits and must be given by a static expression.
2. The specified size is taken as a mandate to store objects of the type in the given size wherever feasible. No attempt is made to store values of the type in a smaller size, even if possible. The following rules apply with regard to feasibility:
   - An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine; that is, no attempt is made to create objects of non-referable size on the stack. If such stack compression is desired, it can be achieved by the user by combining multiple stack variables in a composite object; for example:
     ```
     type My_Enum is (A,B);
     for My_enum'size use 1;
     V,W: My_enum; -- will occupy two storage
                   -- units on the stack
                   -- (if allocated at all)
     type rec is record
        V,W: My_enum;
     end record;
     pragma PACK(rec);
     O: rec;       -- will occupy one storage unit
     ```
   - A formal parameter of the type is sized according to calling conventions rather than size specifications of the type. Appropriate size conversions upon parameter passing take place automatically and are transparent to the user.
   - Adjacent bits to an object that is a component of a composite object, but whose size is non-referable, may be affected by assignments to the object, unless these bits are occupied by other components of the composite object; that is, whenever possible, a component of non-referable size is made referable.

   In all cases, the compiler generates correct code for all operations on objects of the type, even if they are stored with differing representational sizes in different contexts.

   Note: A size specification cannot be used to force a certain size in value operations of the type; for example
   ```
   type my_int is range 0..65535;
   for my_int'size use 16; -- o.k.
   A,B: my_int;
      ...A + B... -- this operation will generally be
                  -- executed on 32-bit values
   ```
3. A size specification for a type specifies the size for objects of this type and of all its subtypes. For components of composite types, whose subtype would allow a shorter representation of the component, no attempt is made to take advantage of such shorter representations. In contrast, for types without a length clause, such components may be represented in a lesser number of bits than the number of bits required to represent all values of the type. For example:
   ```
   type MY_INT is range 0..2**15-1;
   for MY_INT'SIZE use 16; -- (1)
   subtype SMALL_MY_INT is MY_INT range 0..255;
   type R is record
             ...
             X: SMALL_MY_INT;
             ...
           end record;
   ```

5.4.2.5. Specification of Collection Sizes

The specification of a collection size causes the collection to be allocated with the specified size. It is expressed in storage units and need not be static; refer to package `SYSTEM` for the meaning of storage units.

Any attempt to allocate more objects than the collection can hold causes a `STORAGE_ERROR` exception to be raised. Dynamically sized records or arrays may carry hidden administrative storage requirements that must be accounted for as part of the collection size. Moreover, alignment constraints on the type of the allocated objects may make it impossible to use all memory locations of the allocated collection. No matter what the requested object size, the allocator must allocate a minimum of 2 words per object. This lower limit is necessary for administrative overhead in the allocator. For example, a request of 5 words results in an allocation of 5 words; a request of 1 word results in an allocation of 2 words.

In the absence of a specification of a collection size, the collection is extended automatically if more objects are allocated than possible in the collection originally allocated with the compiler-established default size. In this case, `STORAGE_ERROR` is raised only when the available target memory is exhausted. If a collection size of zero is specified, no access collection is allocated.

5.4.2.6. Specification of Task Activation Size

The specification of a task activation size causes the task activation to be allocated with the specified size. It is expressed in storage units; refer to package `SYSTEM` for the meaning of storage units.

Any attempt to exceed the activation size during execution causes a `STORAGE_ERROR` exception to be raised. Unlike collections, there is no extension of task activations.

5.4.2.7. Specification of `'SMALL`

Only powers of 2 are allowed for `'SMALL`.

The length of the representation may be affected by this specification. If a size specification is also given for the type, the size specification takes precedence; it must then be possible to accommodate the specification of `'SMALL` within the specified size.

5.4.3. Enumeration Representation Clauses

For enumeration representation clauses (LRM 13.3), the following restrictions apply:

- The internal codes specified for the literals of the enumeration type may be any integer value between `INTEGER'FIRST` and `INTEGER'LAST`. It is strongly advised to not provide a representation clause that merely duplicates the default mapping of enumeration types, which assigns consecutive numbers in ascending order starting with 0, since unnecessary runtime cost is incurred by such duplication. It should be noted that the use of attributes on enumeration types with user-specified encodings is costly at run time.
- Array types, whose index type is an enumeration type with non-contiguous value encodings, consist of a contiguous sequence of components. Indexing into the array involves a runtime translation of the index value into the corresponding position value of the enumeration type.

5.4.4. Record Representation Clauses

The alignment clause of record representation clauses (LRM 13.4) is observed.

Static objects may be aligned at powers of 2 up to a page boundary. The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record forced by the component allocation and the minimum alignment requirements of the components is already more stringent than the specified alignment.

The component clauses of record representation clauses are allowed only for components and discriminants of statically determinable size. Not all components need to be present. Component clauses for components of variant parts are allowed only if the size of the record type is statically determinable for every variant.

It should be noted that the default type mapping for records maps components of boolean or other types that require only a single bit to a single bit in the record layout, if there are multiple such components in a record. Otherwise, it allocates a referable amount of storage to the component.

If pragma `PACK` is applied to a record for which a record representation clause has been given detailing the allocation of some but not all components, the pragma `PACK` affects only the components whose allocation has not been detailed. Moreover, the strategy of not utilizing gaps between explicitly allocated components still applies.

5.4.7. Minimal Alignment for Types

Certain alignment properties of values of certain types are enforced by the type mapping rules. Any representation specification that cannot be satisfied within these constraints is not obeyed by the compiler and is appropriately diagnosed.

Alignment constraints are caused by properties of the target architecture, most notably by the capability to extract non-aligned component values from composite values in a reasonably efficient manner. Typically, restrictions exist that make extraction of values that cross certain address boundaries very expensive, especially in contexts involving array indexing. Permitting data layouts that require such complicated extractions may impact code quality on a broader scale than merely in the local context of such extractions.

Instead of describing the precise algorithm of establishing the minimal alignment of types, we provide the general rule that is being enforced by the alignment rules:

- No object of scalar type including components or subcomponents of a composite type, may span a target-dependent address boundary that would mandate an extraction of the object's value to be performed by two or more extractions.

5.5. IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record. These components cannot be named by the user.

5.6. INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES

Section 13.5.1 of the Ada Language Reference Manual describes a syntax for associating interrupts with task entries. Tartan Ada implements the address clause

```
for toentry use at intID;
```

by associating the interrupt specified by `intID` with the `toentry` entry of the task containing this address clause. The interpretation of `intID` is both machine and compiler dependent.

The 1750A Ada runtimes provide 16 interrupts that may be associated with task entries. These interrupts are identified by an integer in the range 0..15. The `intID` argument of an address clause is interpreted as follows:

- If the argument is in the range 0..15, a full support interrupt association is made between the interrupt specified by the argument and the task entry.
- If the argument is in the range 16..31, a fast interrupt association is made between the interrupt number (argument-16) and the task entry.
- If the argument is outside the range 0..31, the program is erroneous.

For the difference between full support and fast interrupt handling, refer to Section 8.5.6.

Tartan Ada enforces the restriction that the body of a generic unit must be compiled before the unit can be instantiated. It does not impose the restriction that the specification and body of a generic unit must be provided as part of the same compilation. A recompilation of the body of a generic unit will cause any units that instantiated this generic unit to become obsolete.

5.9.3. Implementation-Defined Characteristics in Package STANDARD

The implementation-dependent characteristics for MIL-STD-1750A in package STANDARD [Annex C] are:

```
package STANDARD is
...
type INTEGER is range -32768 .. 32767;
type FLOAT is digits 6 range -16#0.8000_00#E+32 .. 16#0.7FFF_FF#E+32;
type LONG_INTEGER is range -2147483648 .. 2147483647;
type LONG_FLOAT is digits 9 range -16#0.8000_0000_00#E+32 ..
        16#0.7FFF_FFFF_FF#E+32 ;
type DURATION is delta 0.0001 range -86400.0 .. 86400.0;
        -- DURATION'SMALL = 2#1.0#E-14
...
end STANDARD;
```

5.9.4. Attributes of Type Duration

The type DURATION is defined with the following characteristics:

Attribute	Value
DURATION'DELTA	0.0001 sec
DURATION'SMALL	$6.103516E^{-5}$ sec
DURATION'FIRST	-86400.0 sec
DURATION'LAST	86400.0 sec

5.9.6. Values of Floating-Point Attributes

Attribute	Value for FLOAT
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	16#0.1000_000#E-4 (approximately 9.53674E-07)
SMALL	16#0.8000_000#E-21 (approximately 2.58494E-26)
LARGE	16#0.FFFF_F80#E+21 (approximately 1.93428E+25)
SAFE_EMAX	127
SAFE_SMALL	16#0.1000_000#E-31 (approximately 2.93874E-39)
SAFE_LARGE	16#0.7FFF_FC0#E+32 (approximately 1.70141E+38)
FIRST	-16#0.8000_000#E+32 (approximately -1.70141E+38)
LAST	16#0.7FFF_FF0#E+32 (approximately 1.70141E+38)
MACHINE_RADIX	2
MACHINE_MANTISSA	23
MACHINE_EMAX	127
MACHINE_EMIN	-128
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

5.10. *SUPPORT FOR PACKAGE MACHINE_CODE*

Package MACHINE_CODE provides an interface through which a user may request the generation of 1750A assembly instructions. The Tartan implementation of package MACHINE_CODE is similar to that described in section 13.8 of the Ada LRM, with several added features. Refer to Appendix A of this manual for the specification for package MACHINE_CODE.

5.10.1. *Basic Information*

As required by LRM, Section 13.8, a routine which contains machine code inserts may not have any other kind of statement, and may not contain an exception handler. The only allowed declarative item is a use clause. Comments and pragmas are allowed as usual.

5.10.2. *Instructions*

A machine code insert has the form TYPE_MARK'RECORD_AGGREGATE, where the type must be one of the records defined in package MACHINE_CODE. Package MACHINE_CODE defines three types of records. Each has an opcode and zero, one or two operands. These records allow for the expression of the entire 1750A Assembly language.

5.10.3. *Operands and Address Modes*

An operand consists of a record aggregate which contains the information needed to specify the corresponding Assembly instruction for generation by the compiler.

Each operand in a machine code insert must have an *Address_Mode_Name*. The address modes specified in package MACHINE_CODE are sufficient to provide all address modes supported by the 1750A Assembly language.

In addition, package MACHINE_CODE supplies the address modes Symbolic_Address and Symbolic_Value which allow the user to refer to Ada objects by specifying Object'ADDRESS as the value for the operand. Any Ada object which has the 'ADDRESS attribute may be used in a symbolic operand. Symbolic_Address should be used when the operand is a true address (that is, a branch target for example). Symbolic_Value should be used when the operand is actually a value (that is, one of the source operands of an ADDI instruction).

When an Ada object is used as a *source* operand in an instruction (that is, one from which a value is read), the compiler will generate code which fetches the *value* of the Ada object. When an Ada object is used as the destination operand of an instruction, the compiler will generate code which uses the *address* of the Ada object as the destination of the instruction.

For source operands:

- Symbolic_Address means that the *address* specified by the 'ADDRESS expression is used as the source bits. When the Ada object specified by the 'ADDRESS instruction is bound to a register, this will cause a compile-time error message because it is not possible to "take the address" of a register.
- Symbolic_Value means that the *value* found at the address specified by the 'ADDRESS expression will be used as the source bits. An Ada object which is bound to a register is correct here, because the contents of a register can be expressed on the 1750A.
- PcRel indicates that the *address* of the label will be used as the source bits.
- Any other non-register means that the *value* found at the address specified by the operand will be used as the source bits.

For destination operands:

- Symbolic_Address means that the desired destination for the operation is the *address* specified by the 'ADDRESS expression. An Ada object which is bound to a register is correct here; a register is a legal destination on the 1750A.

In -Mw mode, the compiler does its best to fix any incorrect operands for an instruction but also issues a warning message stating that the machine code insert required additional machine instructions to satisfy the 1750A Assembly language requirements.

5.10.6. Register Usage

Since the compiler may need to allocate registers as temporary storage in machine code routines, there are some restrictions placed on register usage; the compiler automatically frees all of the registers which are volatile across calls for your use (that is, R0, R1, R2, R3). In most instances if you reference any other registers, the compiler reserves them for your use until the end of the machine code routine. However, bear in mind that the compiler does *not* save the registers automatically if the routine is inline expanded. This means that the first reference to a register which is not volatile across calls should be an instruction which saves the register's contents to insure that the value is not overwritten and can later be restored at the end of the routine (by the user). This rule will help ensure correct operation of your machine code inserts even if it is inline expanded by another routine (and possibly another user).

As a result of freeing all volatile registers for the user, any parameters which were passed in registers are moved to either a non volatile register or to memory. References to `Parameter'Address` in machine code inserts will then produce code that access these register or memory locations. This means that there is a possibility of invalidating the value of some `'Address` expressions if the non volatile register to which the value is bound, is referenced as a destination in some later machine code insert. In this case, any subsequent references to the `'Address` expression will cause the compiler to issue a warning message to this effect. In -M mode the compiler uses registers to effect the corrections. If you use all fifteen registers, corrections will not be possible. In general, when more registers are available to the compiler it is better able to generate efficient code.

5.10.7. Data Directives

Three special *instructions* are included in package Machine_Code to allow the user to place a bit pattern into the code stream. These instructions are `DATA16`, `DATA32` and `DATA48`. Each of these instructions can have from 1 to 6 operands.

- `DATA16` places 16-bit values into the code stream. Operands may include integers and addresses.
- `DATA32` places 32-bit values into the code stream. Operands may include integers or floats.
- `DATA48` places 48-bit values into the code stream. The only legal operands are floats.

5.10.8. Honeywell GVSC BIF's

The Honeywell GVSC (Generic VHSIC Spaceborne Computer) version of the 1750A contains some Built-In Functions (BIF's) that are made available in the Machine_Code package. These instructions should be used *only* if the target is the GVSC 1750A. The mnemonics for these instructions are: `SIN, ESIN, COS, ECOS, DMBIF, DMBIF2, BIT, ATAN, EATA, SQRT, ESQR`.

If the *x* and *y*-coordinate arguments of the `ATAN` (or `EATA`) instruction are not known to be in adjacent register pairs (or triples), the Two_Opnds form may be used to specify both coordinates. The first operand is the x-coordinate and the second operand is the *y*-coordinate. Subject to `FIXUP` rules, the compiler will attempt to find adjacent register pairs (or triples) and move the arguments before emitting the instruction.

5.10.9. Inline Expansion

Routines which contain machine code inserts may be inline expanded into the bodies of other routines. This may happen under programmer control through the use of pragma `INLINE`, or at Optimization Level *time* when the compiler selects that optimization as an appropriate action for the given situation. The compiler will treat the machine code insert as though it were a call. For example, volatile registers will be saved and restored around the inline expansion, and similar optimizing steps will be taken.

Consider a procedure that makes a call to an inlined subprogram in the package.

```
with In_Pack;
procedure uses_Inlined_Subp is
begin
   I_Will_Be_Inlined;
end;
```

After the package specification for `In_Pack` has been compiled it is possible to compile the unit `Uses_Inlined_Subp` that makes a call to the subprogram `I_Will_Be_Inlined`. However, because the body of the subprogram is not yet available, the generated code will not have an inlined version of the subprogram. The generated code will use an out of line call for `I_Will_Be_Inlined`. The compiler will issue warning message #2429 that the call was not inlined when `uses_Inlined_Subp` was compiled.

If `In_Pack` is used across libraries, it can be exported as part of a specification library after having compiled the package specification. Note that if only the specification is exported, that in all units in libraries that import `In_Pack` there will be no inlined calls to `In_Pack`. If only the specification is exported, then all calls that appear in other libraries will be out of line calls. The compiler will issue warning message #6601 to indicate the call was not inlined.

There is no warning at link time that subprograms have not been inlined.

If the body for package `In_Pack` has been compiled before the call to `I_Will_Be_Inlined` is compiled, then the compiler will inline the subprogram. In the example above, if the body of `In_Pack` has been compiled before `uses_Inlined_Subp`, then when `uses_Inlined_Subp` is compiled, the call will be inlined.

Having an inlined call to a subprogram makes a unit dependent on the unit that contains the body of the subprogram. In the example, once `uses_Inlined_Subp` has been compiled with an inlined call to `I_Will_Be_Inlined`, the unit `uses_Inlined_Subp` will have a dependency on the package body In_Pack. Thus, if the body for package body In_Pack is recompiled, `uses_Inlined_Subp` will become obsolete, and must be recompiled before it can be linked.

It is possible to export the body for a library unit. If the body for package `In_Pack` is added to the specification library, `exportlib` command, then other libraries that import package `In_Pack` will be able to compile inlined calls across library units.

At optimization levels lower than the default, the compiler will not inline calls, even when pragma `INLINE` has been used and the body of the subprogram is in the library prior to the unit that makes the call. Lower optimization levels avoid any changes in flow of the code that causes movement of code sequences (as happens in a pragma `INLINE`). If the compiler is running at a low optimization level the user will not be warned that inlining is not happening.

```
with intrinsics; use intrinsics;
with int_io;
with text_io;
with flt_io;

procedure Transfer_Words(
   In_Port : integer;
   Out_Port : integer;
   words : integer) is

   function Execute_Output is new XO(integer); -- source has integer type
   function Execute_Input is new XI(integer); -- result has integer type
   a : integer;
begin
   for i in 1..words loop
      a := Execute_Input(PI, In_Port*256); -- CMD = PI+In_Port*256
      Execute_Output(a, PO, Out_Port*256); -- CMD = PI+Out_Port*256
   end loop;
end Transfer_Words;
```

Figure 5-1: Example Use of Intrinsics

```
with intrinsics; use intrinsics;

function shift(
   ShiftMe : integer;
   ShiftCount : integer;
   Signed : boolean) return integer is
   -- Try writing this without these intrinsics!

   function    LogicalShift is new LS(integer,integer);
   function ArithmeticShift is new SA(integer,integer);
begin
   if Signed then return ArithmeticShift(ShiftMe, ShiftCount);
   else           return    LogicalShift(ShiftMe, ShiftCount);
   end if;
end shift;
```

Figure 5-2: LS and SA Used To Define a General Purpose Shift Routine